Universidade Federal
de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

Projeto e Análise de Algoritmos

Trabalho Prático I - KitBOOM

Miguel de Assis Viveiros - 2023008193
Nícolas Rafael Mendonça Teles - 2023008200

Documentação do Trabalho Prático I de Projeto e Análise de Algoritmos. Professor Leonardo Rocha, curso de Ciência da Computação da Universidade Federal de São João Del Rei

São João Del Rei
Novembro de 2024

Sumário

1 Introdução

A empresa Explosivos Jepslon precisava de um programa para embalar corretamente os seus explosivos em uma caixa, o KitBOOM. Possui bombas com 4 polaridades distintas, que são representadas por quatro cores(Amarelo, Verde, Vermelho e Azul), e com tamanhos de 1 centímetro de largura e um comprimento que varia de 1 a 3 centímetros. A caixa tem um tamanho de 6cm x 6cm, e é bem definida, as bombas não podem sofrer movimentos bruscos.

1.1 Descrição do algoritmo

1.1.1 Modo de rodar o programa

O modo esperado de se rodar o programa é o seguinte:

- Primeiro compila-se o programa pelo makefile por meio do comando "make compile"
- Após isso, o executável do programa será gerado. Para rodar o programa em si deve-se seguir a seguinte estrutura:

  - ./main -k [nome_do_arquivo_de_composição].txt -c [nome_do_arquivo_de_configuração].txt
- Caso o usuário deseje rodar um número maior de entradas, para se analisar o tempo de execução do programa, foi adicionado uma terceira opção, -n, para que possa fazer sem que tenha que criar arquivos enormes por conta própria.

  - Essa opção faz com que o arquivo de configuração que foi enviado seja testado diversas vezes, para assim aumentar artificialmente o tamanho da entrada.

  - Caso não seja fornecido um tamanho de entrada, o programa irá assumir que deve testar o arquivo de configuração apenas uma vez.

  - Essa opção recebe como argumento, após o '-n', o número de vezes que o usuário deseja testar o arquivo de configuração.

1.1.2 Lógica do algorimo

Fizemos um programa usando a linguagem C para atender o que a empresa desejava. Inicialmente, o código recebe pelo terminal o nome dos arquivos referente a composição e o de configurações.

Após isso, lê-se, primeiramente, o de composição somando as áreas das bombas e por fim verificando se é compatível com o tamanho da caixa. Caso contrário já encerra o programa e retorna que a configuração está incorreta e também se ela é maior ou menor que a área esperada da caixa.

Se a composição estiver correta, abre o arquivo que possui as configurações que deseja testar, já esperando que todas elas estejam com a mesma composição que fora testada

anteriormente, devido à reserva de espaço que já foi feita. Então o código passa linha por linha do arquivo criando as bombas e posicionando-as em uma matriz de cores do KitBOOM, e ao ler uma linha vazia testa essa configuração, se possui bombas de mesma cor adjacente, caso a configuração esteja correta será impresso no terminal que aquela configuração é válida, caso contrário será impresso que ela é inválida. Isso se repete até o fim do arquivo com as configurações.

2 Descrição das Rotinas

2.1 Lógica

2.1.1 *criaBomba()*

- Descrição: Essa função cria uma estrutura Bomba, alocada dinamicamente, com seu início e fim de linhas e colunas, com o seu tamanho e com sua cor.

- Parâmetros:

  - inicioLinha (Tipo int): é o número da posição inicial da bomba da linha na matriz de posições.

  - inicioColuna (Tipo int): é o número da posição inicial da bomba da coluna na matriz de posições.

  - fimLinha (Tipo int): é o número da posição final da bomba da linha da matriz de posições.

  - fimColuna (Tipo int): é o número da posição inicial da bomba da coluna na matriz de posições.

  - tamanho (Tipo int): é o tamanho da bomba.

  - cor (Tipo Cor): é uma estrutura que salva a cor da bomba em uma string.

- Retorno (Tipo Bomba*):

  - retorna a Bomba alocada dinamicamente e com todos os atributos iniciados corretamente.

2.1.2 *posicionaBomba()*

- Descrição: Essa função salva a bomba que foi recém adicionada no vetor de bombas, buscado-a com o índice que veio como parâmetro, e acessa suas variáveis de posição para posicionar sua cor em sua respectiva localização na matriz.

- Parâmetros:

- KitBooom (Tipo Kit*): é o kit em criação com a configuração atualmente sendo testada. - bombaAdicionada (tipo int): representa a posição no vetor de bombas da bomba que acabou de ser adicionada, e que agora deve ser posicionada na matriz.

- Retorno (Tipo void):

- não é necessário retorno pois os valores necessários são modificados dentro da função por meio de ponteiros.

2.1.3 *testaArquivoComposicao()*

- Descrição: Essa função busca testa a área até o momento calculada de ter sido ocupada do kit e soma tal valor com o valor da área do novo tipo de bomba identificado com o número da mesma que se encontra na composição, para ver se o kit se mantém válido.

- Parâmetros:

- numBombas (Tipo int): é o número do novo tipo de bomba identificado que se encontra listado na composição.

- areaBomba (Tipo int): é a área do novo tipo de bomba identificado que se encontra listado na composição.

- areaTotalBombas (Tipo int*): é a área total até o momento preenchido do kit, passada por endereço para poder ser acessada dentro e fora da própria função múltiplas vezes.

- Retorno (Tipo int):

- -1 (Menor): caso a área necessária do kitBOOM ainda não tenha sido atendida, ainda há espaço sobrando, o que não pode ocorrer.

- 0 (Tamanho correto): caso a área seja exatamente a que deve ser preenchida no kitBOOM.

-1 (Maior): caso a área calculada seja maior do que a área disponível no kit, o que também não pode ocorrer.

2.1.4 *testaArquivoConfiguracao()*

- Descrição: Essa função busca percorrer o vetor de bombas e analisar se possui bombas adjacentes, na matriz de posições, da mesma cor dela.

- Parâmetros:

- KitBooom (Tipo Kit*): é o kit em criação com a configuração atualmente sendo testada.

- Retorno (Tipo int):

  - 0 (inválida): caso essa configuração testada tenha bombas adjacentes de mesma cor na matriz de posições.

  - 1 (válida): caso essa configuração não tenha bombas de mesma cor adjacentes na matriz de posições.

2.1.5 *imprimirMatriz()*

- Descrição: Essa função percorre cada posição da matriz e imprime no terminal a cor que se encontra em tal posição, sendo tudo imprimido na mesma organização da matriz. Primariamente usada para testes.

- Parâmetros:

  - matriz (Tipo Cor**): a matriz com a cor das bombas salva em cada posição.

- Retorno (Tipo void):

  - não é necessário retorno pois é uma função apenas para imprimir a matriz, o que é feito no terminal.

2.1.6 *destroiBomba()*

- Descrição: Essa função libera a estrutura bomba alocada dinamicamente da memória.

- Parâmetros:

  - bomba (Tipo Bomba*): é a bomba alocada dinamicamente.

- Retorno (Tipo void):

  - não é necessário retorno, pois a função só libera da memória a estrutura de dados.

2.1.7 *criaVetorBomba()*

- Descrição: Essa função aloca dinamicamente um vetor com o tamanho de bombas constatadas no arquivo de composição.

- Parâmetros:

  - numeroBombas (Tipo int): é o numero total de bombas do arquivo de composição.

- Retorno (Tipo Bomba**):

  - retorna o vetor de bombas alocado dinamicamente ainda com as bombas nulas.

2.1.8 *criaKit()*

- Descrição: Função que dinamicamente aloca o espaço necessário para os atributos salvos dentro da struct do kit
- Parâmetros:

  - numBombas (Tipo int): número total de bombas do kit, para que seja reservado o espaço necessário para o vetor de bombas
- Retorno (Tipo Kit*):

  - retorna o ponteiro para onde o kit foi alocado.

2.1.9 *destroiMatriz()*

- Descrição: Essa função libera da memória a matriz de Cor alocada dinamicamente.
- Parâmetros:

  - matriz (Cor**): é a matriz de posições da estrutura kit que foi alocada dinamicamente ao criar o Kit.
- Retorno (Tipo void):

  - não é necessário retornar nada, pois é uma função para liberar memória alocada dinamicamente.

2.1.10 *destroiVetorBombas()*

- Descrição: Função que percorre o vetor de bombas por completo, liberando o espaço reservado para cada bomba, e no final libera o espaço reservado para o ponteiro que aponta para o início do vetor.
- Parâmetros:

  - vetor (Tipo Bomba**): vetor de ponteiros que apontam para as bombas alocadas dinamicamente e salvas no kit.

  -n (Tipo int): número total de bombas salvas no vetor de bombas.
- Retorno: (Tipo void)

  - Não é necessário retorno pelo fato de ser apenas uma função que libera a memória que havia sido previamente reservada.

2.1.11 *destroiKit()*

- Descrição: Essa função libera da memória a estrutura Kit alocada dinamicamente, chamando as funções que destrói o vetor de bombas e a matriz.
- Parâmetros:

  - kitBoom (Tipo Kit*): é o kit que foi alocado dinamicamente.
- Retorno (Tipo void):

  - não é necessário retornar nada, pois é uma função para liberar memória alocada dinamicamente.

2.2 Entrada e Saída de Dados

2.2.1 *substituiQuebraDeLinha()*

- Descrição: Substitui o *caracter* de quebra de linha('\n') do final de uma string por um *caracter* de final de string('\0) para evitar erro durante as comparações.
- Parâmetros:

  - string (Tipo char*): string da qual se deseja retirar o *caracter* de quebra de linha.
- Retorno (Tipo void):

  - não retorna nada pois os valores necessários de serem alterados(o \n por \0) são manipulados por ponteiro(char*) dentro da própria função.

2.2.2 *obterNomeArquivos()*

- Descrição: Usando o getopt para conseguir o nome dos arquivos que são utilizados no programa.
- Parâmetros:

  - argc (Tipo int): quantidade de argumentos lidos no terminal.

  - argv (Tipo char**): é um vetor de strings e cada string é um argumento.

  - arquivoKit (Tipo char**): é um ponteiro para a string do arquivo de composição.

  - arquivoConfiguração (Tipo char**): é um ponteiro para a string do arquivo de configuração.
- Retorno (Tipo void):

  - não retorna nada pois todas as variáveis necessárias são modificadas dentro da função por meio de ponteiros.

2.2.3 *lerArquivoComposicao()*

- Descrição: Essa função abre o arquivo de composição e o percorre salvando o número de bombas e calculando sua área total para conferir se é válida para o tamanho do kit.

- Parâmetros:

  - nomeArquivo (Tipo char*): é nome o arquivo de composição.

- Retorno (Tipo Kit*):

  - retorna a estrutura Kit alocada dinamicamente, ou, caso a área do kit seja inválida, interromperá a execução do programa por ali mesmo.

2.2.4 *lerArquivoConfiguracao()*

- Descrição: Essa função abre o arquivo de configuração e o percorre as configurações nele encontradas, criando as bombas e as adicionando no vetor de bombas, assim como posicionando suas cores na matriz de posições, para no final dizer se a configuração testada é valida ou não, e partir para a próxima, caso haja uma.

- Parâmetros:

  - nomeArquivo (Tipo char*): é o nome arquivo de configuração.

  - kitBOOM (Tipo Kit*): o kit alocando dinamicamente com a matriz de posições e o vetor de bombas nulos.

- Retorno (Tipo void):

  - não é necessário retorno na função pois os resultados de validade são escritos no terminal para cada uma das configurações.

2.3 Medição de Tempo

2.3.1 *calculaTempoProcessador()*

- Descrição: Essa função imprime no terminal o tempo do processo no modo de usuário e sistema, bem como o tempo total - sendo esses os dois anteriores somados - e o máximo de memória usada de uma vez por meio da estrutura de dados *rusage*. Também calcula se a soma dos valores dos milissegundos passaria de 1000000, caso no qual subtrai 1000000 do valor e soma um no valor dos segundos

- Parâmetros:

  - usage (Tipo struct rusage): é a estrutura de dados que guarda as informações sobre tempo gasto.

- Retorno (Tipo void):

  - não é necessário retorno na função pois os resultados de tempo gasto são escritos no terminal..

2.3.2 *calculaTempoReal*

- Descrição: Essa função imprime no terminal o tempo que levou para ser executado o código em tempo real em segundos, por meio da subtração do tempo de fim e de início, e no caso de o tempo dos milissegundos do início ser maior que o fim, faz a diferença e então subtrai um do valor dos segundos.

- Parâmetros:

  - start (Tipo struct timeval): Estrutura que salva o tempo de início.

  -end (Tipo struct timeval): Estrutura que salva o tempo de término.

- Retorno (Tipo void):

  - não é necessário retorno na função pois o tempo é impresso no terminal.

3 Descrição das Estruturas de Dados

3.1 Cor

- Descrição: Essa estrutura de dados serve para salvar as cores das bombas e facilitar o manuseio da matriz de posições do Kit.

- Atributos:

  - cor (Tipo char[3]): é uma string que terá uma string de 3 posições com a cor da bomba(2 letras+`\0`).

3.2 Bomba

- Descrição: Essa estrutura de dados salva os dados de uma bomba, como início e fim nas linhas e colunas, seu tamanho e sua cor.

- Atributos:

  - inicioLinha (Tipo int): é o número da posição inicial da bomba da linha na matriz de posições.

  - inicioColuna (Tipo int): é o número da posição inicial da bomba da coluna na matriz de posições.

  - fimLinha (Tipo int): é o número da posição final da bomba da linha da matriz de posições.

  - fimColuna (Tipo int): é o número da posição inicial da bomba da coluna na matriz de posições.

  - tamanho (Tipo int): é o tamanho da bomba.

  - cor (Tipo Cor): é onde salva a cor da bomba.

3.3 Kit

- Descrição: Essa estrutura de dados da caixa de bombas que é o ponto central do algoritmo. Caixa essa que contém salva nela a quantidade de bombas existentes, bem como as configurações de cada uma delas Além disso temos uma matriz do tipo cor, para salvar as posições na caixa ocupadas por cada bomba de cada cor.

- Atributos:

  - numBomba (Tipo int): é o número de bombas constatada no arquivo de composição.

  - matrizPosicoes (Tipo Cor**): matriz de posições de onde cada bomba ficaria na caixa.

- vetorBombas (Tipo Bomba**): é o vetor que possui todas as bombas, da configuração atualmente analisada.

4 Análise de complexidade das rotinas

Iremos fazer a análise de complexidade das rotinas de nosso algoritmo, considerando uma operação essencial num modelo matemático específico para cada uma das rotinas. Também iremos definir que nossa variável n, o tamanho da entrada, será definida pelo número de linhas do nosso arquivo de configurações, e a variável m, definida pelo número de linhas(ou seja, bombas) por configuração de caixa, e uma última variável, p, sendo essa o número de configurações que se deseja testar.

4.1 Lógica

4.1.1 *criaBomba()*

Nessa função o modelo matemático definido levará em conta a alocação da bomba na memória e inicializar os seus atributos. Levando isso em consideração temos 7 operações, ou seja, f(n, m, p) = 7, já que o número de linhas de nenhum dos arquivos mudará a quantidade de operações que foram consideradas.

4.1.2 *posicionaBomba()*

Nessa função o modelo matemático definido será a copia de strings na matriz de posições. No pior caso será feita 3 copias de strings, já que o tamanho máximo das bombas é 3, cada uma dessas strings com 3 posições. Com isso a função será f(n, m, p) = 3*3=9.

4.1.3 *testaArquivoComposicao()*

O modelo matemático dessa função irá levar como operação essencial seria fazer a comparação da nova área ocupada no kit, com a área esperada, portanto no pior dos casos essa função executará 2 comparações, f(n, m, p) = 2.

4.1.4 *testaArquivoConfiguracao()*

A operação essencial considerada no modelo matemático dessa rotina será a comparação com as posições adjacentes, portanto no pior caso, em que a caixa está completamente preenchida por bombas de 1x1, já que nesse caso é necessário sempre comparar com todas as posições adjacentes, não há posições ignoradas por ser parte da mesma bomba. O número de comparações pode ser pensado da seguinte maneira. As 4 bombas nas pontas fariam 2 comparações, 4*2=8 comparações. As outras 4 bombas em cada 2 linhas(topo e baixo) e 2 colunas(direita e esquerda) executarão 3 comparações, portanto 3*4*(2+2)=3*4*4=48

comparações. Já foram consideradas 4 pontas + 4 em cada borda, 4+4*4=20 bombas, as outras 6*6-20=36-20=16 bombas executarão 4 comparações cada, pois terão que comparar com todas as posições adjacentes, encima, embaixo, direita e esquerda. Então essas bombas restantes executarão 16*4=64 comparações no total. Isso significa que o total de comparações no kit completo nesse pior caso que está sendo considerado seria de 64+48+8=120 comparações. Fazendo a média de quantas comparações cada bomba faria, $120/36 \approx 3.33$ comparações por bomba. Levando em consideração o fato que o número de linhas por configuração(n) seria o número de bombas de dita configuração, podemos dizer que, no pior caso, teremos 3.33n comparações em cada configuração sendo testada. Portando a função de complexidade dessa rotina se encontra da seguinte maneira: f(n, m, p)=3.33n.

4.1.5 *imprimirMatriz()*

Nessa função o modelo matemático será a quantidade de vezes que será printado no terminal. Sabendo que o tamanho da matriz é de 6 por 6, temos 36 prints no terminal para representar a matriz, porém também há o print de quebra de linha que será feito 6 vezes, logo temos no total 42 chamadas do printf. A função então é f(n, m, p) = 42.

4.1.6 *destroiBomba()*

Nessa função o modelo matemático será a quantidade de vezes que chama a função free. Como é chamada somente uma única vez temos que f(n, m, p) = 1.

4.1.7 *criaVetorBomba()*

Definiremos o modelo matemático dessa função para levar em consideração a operação de reserva de memória, que é executada apenas uma vez por chamada de função, então a função de complexidade dessa função será constante em f(n, m, p) = 1.

4.1.8 *criaKit()*

Nessa função o modelo matemático será considerado inicialização dos atributos do Kit e sua alocação na memória. É feita 10 operações, considerando que é feita 6 vezes no loop que inicializa as linhas da matriz. Sendo assim a função é f(n, m, p) = 10.

4.1.9 *destroiMatriz()*

O modelo matemático dessa função irá levar em consideração a operação de liberação de memória. Essa operação é executada 6 vezes, para liberar o espaço de cada linha da matriz, depois mais uma para liberar o espaço da matriz em si, portando a função de

complexidade será constante, não variando com número de linhas dos arquivos, resultando em f(n, m, p) = 7.

4.1.10 *destroiVetorBombas()*

O modelo matemático considerado nessa função irá ser calculado com relação às operações de liberação de memória, que primeiro executará n iterações da função *destroiBomba()* - a qual calculamos complexidade f(n, m, p)=1 - uma para cada bomba do kit, e no final uma última liberação, para liberar o espaço reservado para o ponteiro do vetor em si. Isso resultará na seguinte função de complexidade final f(n, m, p)=n*1+1=n+1.

4.1.11 *destroiKit()*

Nessa função o modelo matemático considerado será a liberação da memória. Já fora calculado a complexidade das funções de destruir, que são funções de liberar memória, da matriz e o vetor de bombas, que são, respectivamente, f(n, m, p) = 7 e f(n, m, p)=n+1, mais o free de liberar a estrutura. Sabendo disso temos que f(n, m, p)=n+1+7+1= n+9.

4.2 Entrada e Saída de Dados

4.2.1 *substituiQuebraDeLinha()*

Nessa função consideraremos um modelo matemático que leva como operação essencial a comparação dos caracteres. Levando em consideração isso, e o fato que cada linha do arquivo de configuração tem 12 caracteres(contando com o `\n`), portanto a função de complexidade dessa rotina seria no pior caso definida por f(n, m, p) = 12, executando 12 comparações por linha quando analisando o arquivo de configuração, enquanto o arquivo de composição possui 7 caracteres por linha contando com o `\n`, portanto seria f(n, m, p) = 7 no caso de estar analisando o arquivo de composição.

4.2.2 *obterNomeArquivos()*

Nessa função o modelo matemático definido levará em conta o número de comparações no switch-case. Será feita 3 comparações no total, já que será somente comparado o -k e o -c, sendo que em um será feita duas comparações e no outro será feito somente uma que já estará certa. Com isso a sua função de complexidade será de f(n, m, p) = 3, já que o números de linhas no arquivo de composição ou no de configuração não alterará a quantidade de comparações feitas.

4.2.3 *lerArquivoComposicao()*

Nessa função o modelo matemático considerado será as atribuições e as chamadas de funções do '*prototipo.h*'. Sendo assim na primeira iteração será feita 3 atribuições,

mais uma chamada de substituir quebra de linha e uma chamada do testa arquivo de composição, que possuem complexidade igual a, respectivamente, f(n, m, p) = 7 e f(n, m, p) = 2. Temos que em uma iteração terá f(n, m, p) = 7+2+3 = 12, logo a função final será f(n, m, p) = 12*m, já que será feita essas operações o por cada linha do arquivo de composição.

4.2.4 *lerArquivoConfiguracao()*

No modelo matemático dessa rotina levaremos em consideração a operação de atribuição dos valores às variáveis do kit e as chamadas de funções internas, as quais já calculamos suas complexidades. Para cada linha do arquivo, temos que são atribuídas as variáveis de inicioColuna, inicioLinha, fimColuna, fimLinha, tamanho, e 3 atribuições para cor, uma em cada posição da string. Além disso ocorre a chamada de função *criabomba()*, que calculamos complexidade de f(n, m, p)=7 e sua atribuição, mais a chamada de posicionaBomba, que calculamos uma complexidade de f(n, m, p)=9. Até o momento temos uma complexidade de 5(atribuições ints)+3(atribuições string)+7(*criabomba()*) + 9(*posicionaBomba()*) = 24 operações por bomba(n) por configuração(p).

Além disso temos operações que são feitas apenas uma vez por configuração, que iremos levar em consideração agora: temos a chamada da função *testaArquivoConfiguracao()* que calculamos uma complexidade de f(n, m, p)=3.33n, a chamada da função *destroiVetorBombas()* com uma complexidade de n+1 e a função *criaVetorBomba()* com complexidade de 1 e uma atribuição para salvar o endereço desse novo vetor de bombas. Então calculamos agora 3.33n+n+1+1+1=4.33n+3, isso para cada configuração em nosso arquivo.

Agora calculando a complexidade total dessa rotina temos: complexidade primeiro encontrada vezes bombas vezes configurações, isso somado com a segunda complexidade encontrada vezes configurações, da seguinte maneira: f(n, m, p)=24*n*p + 4.33n+4*p = p*(24n+4.33n+3) => f(n, m, p)=(28.33n+3)*p

4.3 Casos de teste

4.3.1 *casoTesteComposicao()*

O modelo matemático dessa função levará em consideração a chamada de função que ocorre dentro da mesma(*lerArquivoComposicao()*), e como tal operação apenas ocorre uma vez, podemos dizer que suas funções de complexidades serão iguais, portanto: f(n, m, p)=12m

4.3.2 *casoTesteConfiguracao()*

O modelo matemático dessa função levará em consideração a chamada de função que ocorre dentro da mesma(*lerArquivoConfiguracao()*), e como tal operação apenas ocorre

uma vez, podemos dizer que suas funções de complexidades serão iguais, portanto: f(n, m, p)=(28.33+3)p

4.4 Medições de tempo

4.4.1 *calculaTempoProcessador()*

O modelo matemático dessa função irá levar em conta as operações aritméticas feitas com os valores e a comparação com o valor de 1000000 para saber se é necessário alterar o valor da variável dos segundos inteiros. Sabendo disso, vemos que há, no caso da comparação ser verdadeira - que seria o pior caso - 4 operações aritméticas e uma de comparação, tal valor não variando com tamanho da entrada, portanto a função de complexidade da rotina é: f(n, m, p)=5

4.4.2 *calculaTempoReal*

As operações essenciais consideradas no modelo matemático dessa rotina serão as operações aritméticas feitas com os valores e a comparação com o valor de 1000000 para saber se é necessário alterar o valor da variável dos segundos inteiros. Levando em consideração o caso da comparação ser verdadeira, que geraria o maior número de operações, temos 4 operações aritméticas e uma comparação, totalizando uma função de complexidade de f(n, m, p)=5, já que o número de operações não se altera com tamanho da entrada.

4.5 Main

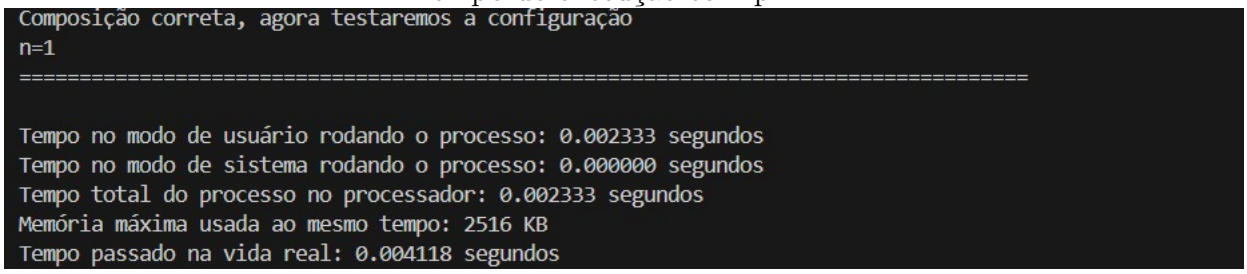
O modelo matemático criado para a função main levará em consideração as chamadas das outras funções com as complexidades previamente estabelecidas, assim calculando a complexidade do algoritmo por completo. Começamos com uma chamada da função *obterNomeArquivos()*, que definimos ter a complexidade de f(n, m, p)=3. Após isso uma chamada da função *lerArquivoComposicao()* com complexidade f(n, m, p)=12m. Em seguida chamada da função *lerArquivoConfiguracao()*, que calculamos a complexidade de f(n, m, p)=(28.33n+3)p. Após isso, temos a chamada das 2 funções de cálculo de tempo, cada uma com uma complexidade calculada de f(n, m ,p)=5.

Como as funções são apenas chamadas uma após a outra iremos apenas somar suas funções de complexidade e encontrar a função da execução completa do código em função dos tamanhos de arquivos. Assim temos: (28.33n+3)p + 12m + 3 + 5 + 5 => f(n, m, p) = (28.33n+3)p + 12m + 13 como a função de complexidade do nosso algoritmo.

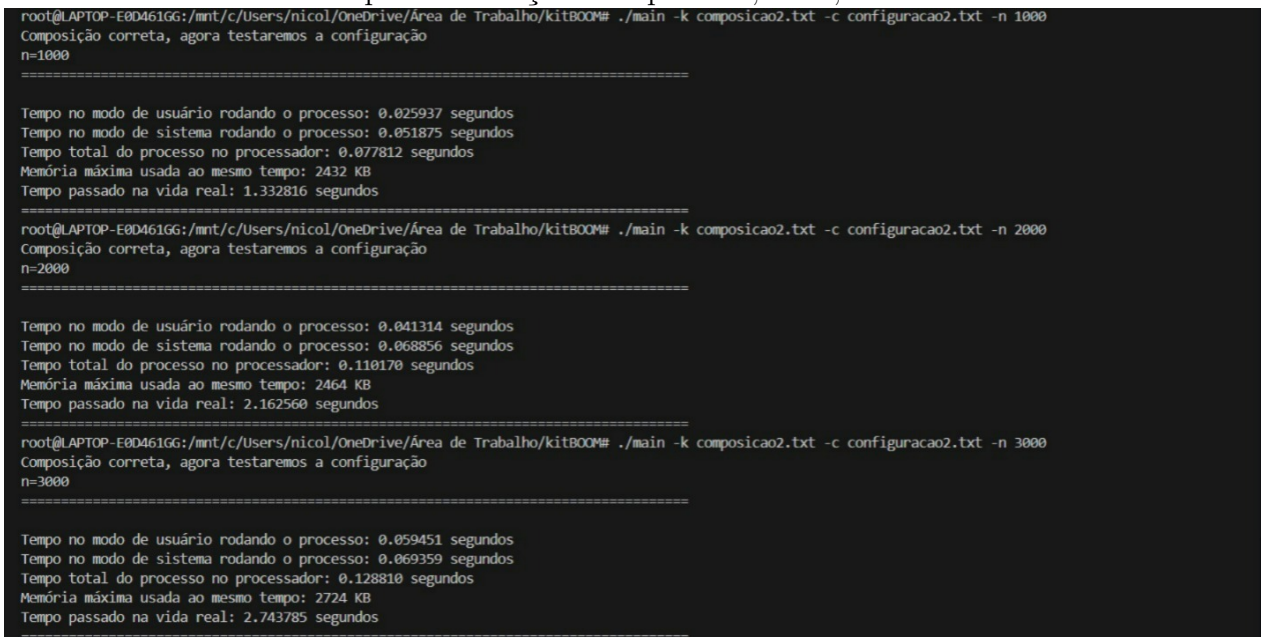
5 Análise dos resultados obtidos

5.1 Imagens demonstrando os tempos de execução e os respectivos tamanhos de arquivos de entrada

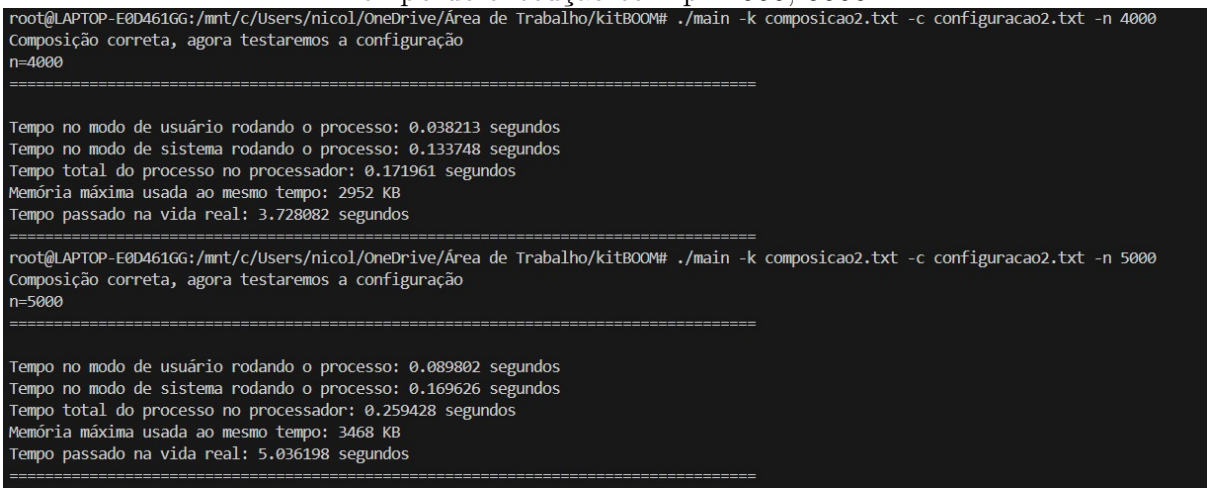
Tempo de execução com p=1



Tempo de execução com p=1000, 2000, 3000



Tempo de execução com p=4000, 5000



Tempo de execução com p=6000, 7000, 8000

```
root@LAPTOP-E0D461GG:/mnt/c/Users/nicol/OneDrive/Área de Trabalho/kitBOOM# ./main -k composicao2.txt -c configuracao2.txt -n 6000
Composição correta, agora testaremos a configuração
n=6000
================================================================================

Tempo no modo de usuário rodando o processo: 0.110368 segundos
Tempo no modo de sistema rodando o processo: 0.171683 segundos
Tempo total do processo no processador: 0.282051 segundos
Memória máxima usada ao mesmo tempo: 3736 KB
Tempo passado na vida real: 5.790370 segundos
================================================================================
root@LAPTOP-E0D461GG:/mnt/c/Users/nicol/OneDrive/Área de Trabalho/kitBOOM# ./main -k composicao2.txt -c configuracao2.txt -n 7000
Composição correta, agora testaremos a configuração
n=7000
================================================================================

Tempo no modo de usuário rodando o processo: 0.111252 segundos
Tempo no modo de sistema rodando o processo: 0.234867 segundos
Tempo total do processo no processador: 0.346119 segundos
Memória máxima usada ao mesmo tempo: 4320 KB
Tempo passado na vida real: 6.993519 segundos
================================================================================
root@LAPTOP-E0D461GG:/mnt/c/Users/nicol/OneDrive/Área de Trabalho/kitBOOM# ./main -k composicao2.txt -c configuracao2.txt -n 8000
Composição correta, agora testaremos a configuração
n=8000
================================================================================

Tempo no modo de usuário rodando o processo: 0.094761 segundos
Tempo no modo de sistema rodando o processo: 0.338435 segundos
Tempo total do processo no processador: 0.433196 segundos
Memória máxima usada ao mesmo tempo: 4472 KB
Tempo passado na vida real: 8.771392 segundos
```

Tempo de execução com p=9000, 10000

```
root@LAPTOP-E0D461GG:/mnt/c/Users/nicol/OneDrive/Área de Trabalho/kitBOOM# ./main -k composicao2.txt -c configuracao2.txt -n 9000
Composição correta, agora testaremos a configuração
n=9000
================================================================================

Tempo no modo de usuário rodando o processo: 0.115504 segundos
Tempo no modo de sistema rodando o processo: 0.404264 segundos
Tempo total do processo no processador: 0.519768 segundos
Memória máxima usada ao mesmo tempo: 5048 KB
Tempo passado na vida real: 10.345607 segundos
================================================================================
root@LAPTOP-E0D461GG:/mnt/c/Users/nicol/OneDrive/Área de Trabalho/kitBOOM# ./main -k composicao2.txt -c configuracao2.txt -n 10000
Composição correta, agora testaremos a configuração
n=10000
================================================================================

Tempo no modo de usuário rodando o processo: 0.139508 segundos
Tempo no modo de sistema rodando o processo: 0.406899 segundos
Tempo total do processo no processador: 0.546407 segundos
Memória máxima usada ao mesmo tempo: 5208 KB
Tempo passado na vida real: 11.301480 segundos
```

5.2 Métodos da análise

Para a análise do tempo de execução do algoritmo, foram dispostos num gráfico 11 execuções, as 11 representadas nas seção anterior, com o tamanho da entrada disposto no eixo x, e o tempo de execução disposto no eixo y.

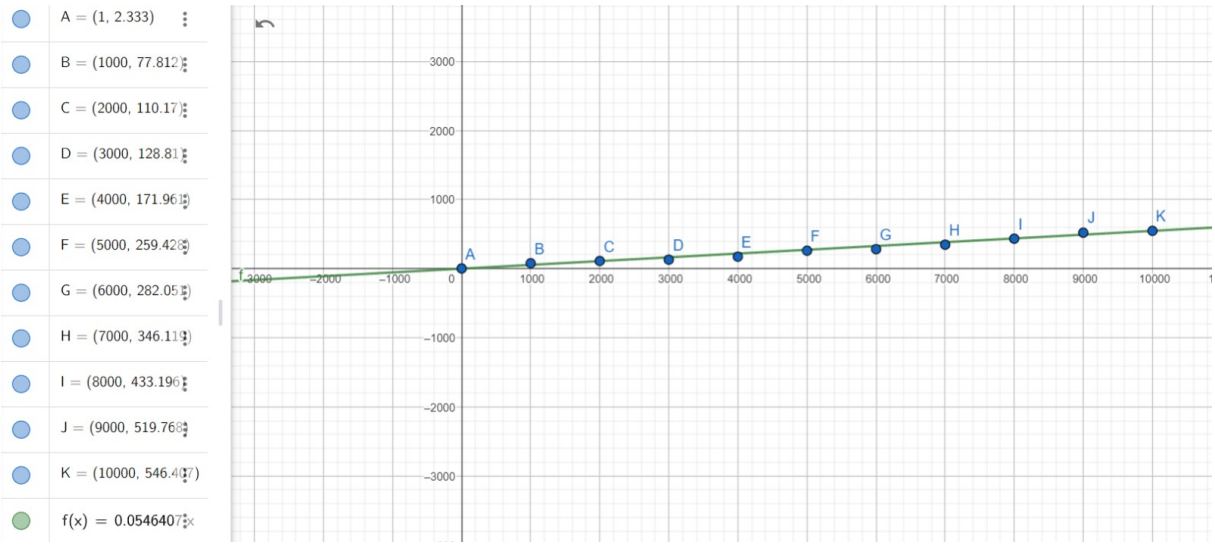
O tempo de execução foi disposto no gráfico em milissegundos, para facilitar sua representação, já que se fossem dispostos em segundos, ficaria difícil de observar o gráfico, pois os valores não variariam o suficiente no eixo y para serem perceptíveis a olho nu. Além disso, o tempo representado no gráfico é o tempo total do processador - tempo de usuário e sistema somados - para que seja uma análise mais precisa da complexidade, invés de se utilizar a medição menos precisa de tempo passado na vida real, já que nem todo o tempo que o processo está ativo em tempo real ele está sendo executado pelo processador.

Para gerar o gráfico dos tempos de execução com arquivos de tamanhos grandes, que seriam inviáveis de serem criados à mão, foi feito o seguinte: Utilizou-se um *for-loop*, que recebia um dos argumentos da linha de comando caso o usuário desejasse rodar o arquivo mais de uma vez, e caso não fosse passado o argumento assumia-se que era apenas uma vez que deve ser rodado. Usamos esse *for-loop* então, para rodar o código com entradas variando em vários tamanhos, porém variamos apenas a variável 'p' em nossos testes.

Essa escolha foi feita pois, os valores de tamanho de entrada que podem variar são: número de linhas do arquivo de composição, número de bombas por configuração e número de configurações no arquivo. Como o funcionamento de nosso código assume que o número de bombas por configuração, uma vez atribuído, não pode mudar, não poderíamos mudar a quantidade de bombas por configuração(n) no meio da execução. Além disso como apenas uma composição é testada, e para ela várias configurações, não há como mudar a quantidade de linhas no arquivo de composição(m) durante a execução.

5.3 A análise

Aqui podemos ver o gráfico representando o tempo de execução por tamanho de entrada.



Nesse gráfico pode-se observar que, conforme variamos a quantidade de configurações testadas, o tempo de execução varia na proporção esperada segundo a complexidade calculada - f(n, m, p)=(28.33n+3)p + 12m +13 - variando de maneira relativamente linear, e já que as outras variáveis mantemos constantes e p variamos, a maneira que ele deveria alterar o número de operações e o tempo de execução é exatamente esse: linear. Portanto aqui demonstramos a representação real de nossa função de complexidade testada pelo tempo de execução do nosso código.